

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Method and System for Allowing Code To Be Securely
Initialized in a Computer**

Inventor(s):

**Paul England
Bryan Willman**

ATTORNEY'S DOCKET NO. MS1-654US

1 **TECHNICAL FIELD**

2 This invention relates to secure code execution, and more particularly to a
3 method and system for allowing code to be securely initialized in a computer.
4

5 **BACKGROUND OF THE INVENTION**

6 Having people be able to trust computers has become an increasingly
7 important goal. This trust generally focuses on the ability to trust the computer to
8 use the information it stores or receives correctly. Exactly what this trust entails
9 can vary based on the circumstances. For example, multimedia content providers
10 would like to be able to trust computers to not improperly copy their content. By
11 way of another example, users would like to be able to trust their computers to
12 forward confidential financial information (e.g., bank account numbers) only to
13 appropriate destinations (e.g., allow the information to be passed to their bank, but
14 nowhere else). Unfortunately, given the generally open nature of most computers,
15 a wide range of applications can be run on most current computers without the
16 user's knowledge, and these applications can compromise this trust (e.g., forward
17 the user's financial information to some other destination for malicious use).

18 To address these trust issues, different mechanisms have been proposed
19 (and new mechanisms are being developed) that allow a computer or portions
20 thereof to be trusted. Generally, these mechanisms entail some sort of
21 authentication procedure where the computer can authenticate or certify that at
22 least a portion of it (e.g., certain areas of memory, certain applications, etc.) are at
23 least as trustworthy as they present themselves to be (e.g., that the computer or
24 application actually is what it claims to be). In other words, these mechanisms
25 prevent a malicious application from impersonating another application (or

allowing a computer to impersonate another computer). Once such a mechanism can be established, the user or others (e.g., content providers) can make a judgment as to whether or not to accept a particular application as trustworthy (e.g., a multimedia content provider may accept a particular application as being trustworthy, once the computer can certify to the content provider's satisfaction that the particular application is the application it claims to be). However, installing such mechanisms on a computer can be difficult, as they require protection against a malicious application interfering with the mechanism (e.g., a malicious application impersonating the trusted mechanism).

One solution is to build a computer that includes a trustworthy mechanism for booting the computer. However, booting a computer typically involves using various pieces of code, often referred to as the basic input output system (BIOS) and potentially many option read only memories (ROMs) or BIOS extensions, that operate to load the operating system from some other storage device (typically a hard disk drive). Thus, a trustworthy mechanism for booting the computer would require that the BIOS be trustworthy, each of the option ROMs be trustworthy, and each of the BIOS extensions be trustworthy, before a trustworthy operating system can be loaded into the computer. Not only is it difficult to have each of these components trustworthy, but the BIOS, option ROMs, and BIOS extensions are frequently stored on devices that can be re-written in the computer (e.g., Flash memory), and thus the integrity thereof compromised. Therefore, building a computer that includes a trustworthy mechanism for booting the computer can be problematic.

The invention described below addresses these disadvantages, providing a method and system for allowing code to be securely initialized in a computer.

1
2 **SUMMARY OF THE INVENTION**

3 A method and system for allowing code to be securely initialized in a
4 computer is described herein.

5 According to one aspect, a memory controller prevents CPUs and other I/O
6 bus masters from accessing memory during a code (for example, OS, microkernel,
7 or other trusted core) initialization process. The memory controller resets CPUs in
8 the computer and allows a CPU to begin accessing memory at a particular location
9 (identified to the CPU by the memory controller). Once an initialization process
10 has been executed by that CPU, the code is operational and any other CPUs are
11 allowed to access memory (after being reset), as are any other bus masters (subject
12 to any controls imposed by the initiated code).
13

14 **BRIEF DESCRIPTION OF THE DRAWINGS**

15 The present invention is illustrated by way of example and not limitation in
16 the figures of the accompanying drawings. The same numbers are used
17 throughout the figures to reference like components and/or features.

18 Fig. 1 is a block diagram illustrating an exemplary computer in accordance
19 with certain embodiments of the invention.

20 Figs. 2 and 3 illustrate exemplary manners in which a trusted core can be
21 implemented.

22 Fig. 4 illustrates an exemplary computer architecture employing distributed
23 memory controllers in accordance with certain embodiments of the invention.

24 Fig. 5 is a flowchart illustrating an exemplary process for a code
25 initialization process in accordance with certain embodiments of the invention.

DETAILED DESCRIPTION

A method and system for allowing code (e.g., software instructions) to be securely initialized in a computer is described herein. This secure code initialization process is described primarily with reference to initializing a trusted core in a computer. However, the secure code initialization process can be used to securely initialize any of a wide variety of types of code, without regard for how trustworthy the code is. The mechanism described herein is of particular interest because no changes to the core-CPU, and few changes to the CPU supporting logic, are required.

As used herein, code being "trusted" refers to code that is immutable in nature and immutable in identity. Code that is trusted is immune to being tampered with by other parts (e.g. code) of the computer and it can be reliably and unambiguously identified. In other words, any other entity or component asking "who is this code" can be told "this is code xyz", and can be assured both that the code is indeed code xyz (rather than some imposter) and that code xyz is unadulterated. Trust does not deal with any quality or usefulness aspects of the code – only immutability of nature and immutability of identity. The method and system described herein can be used to allow trusted code to know that it started execution in an orderly state, providing immutability of initialization.

Fig. 1 is a block diagram illustrating an exemplary computer 100 in accordance with certain embodiments of the invention. Computer 100 is intended to represent a wide variety of computing devices, such as personal computers (PCs), hand-held or pocket devices, personal digital assistants (PDAs), gaming consoles, Internet appliances, multiprocessor or single-processor systems,

1 microprocessor-based or programmable consumer electronics, network PCs,
2 minicomputers, mainframe computers, embedded systems, etc.

3 Computer 100 includes one or more central processing units (CPUs) or
4 processors 102, 104, a memory controller 106, an I/O controller 108, and system
5 memory 110. CPUs 102 and 104 represent any of a wide variety of processors,
6 such as x86-architecture (or compatible) processors, PowerPC-architecture (or
7 compatible) processors, etc. Although multiple (n) CPUs 102, 104 are illustrated
8 in Fig. 1, computer 100 can alternatively be implemented with only a single CPU.
9 The CPUs 102, 104 are coupled together and to memory controller 106 via a
10 processor bus 112.

11 Memory controller 106 is illustrated as a separate component, but may
12 alternatively be incorporated into another component or device, such as a bridge
13 including an interrupt controller, accelerated graphics port (AGP), etc. Memory
14 controller 106 communicates with CPUs 102, 104, I/O controller 108, and
15 memory 110. Memory 110 represents a system memory, such as a random access
16 memory (RAM). I/O controller 108 operates as a bridge or interface between
17 various I/O devices coupled to an I/O bus 114 and other components within
18 computer 100. Any of a wide variety of conventional I/O devices can be coupled
19 to I/O bus 114. In the illustrated example, a mass storage device controller 116 is
20 coupled to bus 114 and a mass storage device (e.g., a hard disk) 116, a ROM 120
21 (including a BIOS 122 and option ROMs and/or BIOS extensions 124) is coupled
22 to bus 114, and an adapter controller 126 is coupled to bus 114 and a network
23 adapter 128 (e.g., network interface card (NIC) or modem).

24 Memory controller 106 includes a processor bus interface 130, a memory
25 interface 132, an I/O interface 134, a control logic (also referred to as a processor

or controller) 136, a digest 138, two reset vectors 140, 142, and initialization parameters 153. Processor bus interface 130 operates as an interface between memory controller 106 and processor bus 112, allowing memory controller 106 to receive commands and requests from CPUs 102, 104 as well as communicate responses and commands to CPUs 102, 104. Memory interface 132 operates as an interface to system memory 110, allowing controller 106 to read data stored in memory 110 and write data to memory 110. I/O interface 134 operates as an interface to I/O controller 108, allowing I/O components 118, 120, and 128 to communicate with memory controller 106 and vice versa via I/O controller 108. In the illustrated example, communications to and/or from a component internal to controller 106 (e.g., control logic 136, reset vectors 140, 142, etc.) are passed through the appropriate interface.

Selected ones of I/O components coupled to I/O bus 114 may operate as bus masters. A bus master refers to a device (typically other than a CPU) that is capable of driving address bus signals and other bus control signals on a bus. Examples of bus masters are mass storage device controller 116 and network adapter controller 126. Bus masters can communicate with memory 110 via I/O controller 108 and 106, such as to perform direct memory access (DMA) transfers to and/or from memory 110, thereby alleviating a CPU 102, 104 from having to directly manage such transfers.

In the illustrated example of Fig. 1, I/O controller 108 communicates directly with memory controller 106. Alternatively, these communications may not pass through memory controller 106, but rather through a processor to I/O bridge (not shown). Additionally, a processor to I/O bridge also allows for communications between I/O controller 108 and CPUs 102, 104 on processor bus

1 112. In one implementation, memory controller 106 is included as part of a
2 "chipset" that also includes such a processor to I/O bridge. A chipset refers to a
3 set of one or more chips that provides the interfaces between the computer's
4 subsystems, and includes the buses and electronics to allow the CPUs, memory,
5 I/O devices, etc. to interact.

6 Control logic 136 operates to ensure a trusted core can be loaded securely
7 in computer 100, allowing accesses to memory 110 only under specific
8 circumstances, as discussed in more detail below. Control logic 136 can be
9 implemented in any of a variety of manners, such as a microcontroller executing
10 microcode (stored in a nonvolatile secure location (e.g., within controller 106)
11 accessible only to the microcontroller), a programmable logic device or other
12 application specific integrated circuit (ASIC), etc.

13 Upon starting-up (e.g., powering-on or resetting) computer 100, the state of
14 memory 110 is typically not guaranteed. Memory 110 is typically a dynamic
15 RAM (DRAM) which requires periodic refreshing in order to maintain its state, so
16 any state would have been lost when computer 100 was not powered. Memory
17 110 can be configured to be at a particular state (e.g., storing all zeroes or all ones)
18 at power-on, or alternatively memory 110 may store random values (e.g., whatever
19 values the individual memory cells stabilized to during power-on). Regardless,
20 memory 110 stores no reliable data or instructions upon power-on. It should be
21 noted that the contents of memory are more or less random when starting-up the
22 entire computer system or just the memory. However, power-up and/or resetting
23 of the CPU(s) do not typically alter the state of the memory (so the contents of
24 memory are not necessarily random after simply resetting a CPU).
25

After computer 100 has been powered-on, one of CPUs 102, 104 will eventually begin executing instructions. Only one of CPUs 102, 104 may initially begin executing instructions, or alternatively one of CPUs 102, 104 may be selected to begin booting the computer. Any of a wide variety of arbitration mechanisms can be used to determine which of multiple CPUs is to begin booting the computer, such as a pre-configured default processor, a distributed arbitration scheme, an arbitration device or component (not shown) coupled to processor bus 112, etc.

Regardless of how many CPUs or which CPU begins execution of instructions, at least one CPU does begin executing instructions. As computer 100 has just been powered-on, no instructions are stored in the CPU's cache memory (or any other cache memory that may be present in computer 100). The CPU obtains instructions from BIOS 122 and begins executing those instructions. In one implementation, the CPU is configured to initially place a read request for a particular address (e.g., FFFFFFFF₁₆) on processor bus 112. Memory controller 106 responds with the address of a BIOS boot block (an address in BIOS 122) at which the CPU should begin executing instructions. This address returned by memory controller 112 is referred to as the reset vector, and in this (common) example, is provided by external logic. In an alternative implementation, the CPU simply begins fetching instructions directly from a particular location (for example, 00000000₁₆), so in this case the reset vector is fixed.

The CPU then begins loading and executing instructions starting with this reset vector. The instructions loaded are from ROM 120, and can include instructions from BIOS 122 as well as one or more option ROMs and/or BIOS extensions 124. Some instructions from ROM 120 will result in instructions being

1 loaded into memory 110 (either from ROM 120 or another device, such as mass
2 storage device 118) that are then loaded by the CPU and executed.

3 Various components 144 of an operating system are thus loaded into
4 memory 110, from ROM 120 and/or other devices (e.g., storage device 118). The
5 components 144 can include portions of BIOS 122 as well as option ROMs and
6 BIOS extensions 124. OS components 144 may also include selected components
7 of a general purpose operating system, such as any of the Windows® family of
8 operating systems available from Microsoft Corp. of Redmond, WA.

9 Eventually, a trusted core is loaded into memory 110. This can be in
10 response to, for example, instructions in BIOS 122, an option ROM or BIOS
11 extension 124, an OS component 144, or another application (not shown). A
12 trusted core refers to trustworthy code (e.g., software instructions) that controls at
13 least a portion of the operation of computer 100, and knows how to protect itself
14 (at least to some extent) on the particular hardware of computer 100. In protecting
15 itself, the trusted core is also able to protect other applications running under the
16 control of the trusted core. The amount of protection provided by the trusted core,
17 as well as the manner in which such protection is provided by the trusted core, can
18 vary among computers, computer (or CPU) architectures, and trusted cores. The
19 trusted core may include all of the functionality of a typical operating system, or
20 alternatively only selected functionality. The trusted core is typically
21 cryptographically certified in some manner, allowing the integrity of the trusted
22 core to be verified at a later time. For example, a cryptographic measure of the
23 trusted core may be extracted from the trusted core and stored along with the
24 trusted core on a mass storage device 118 (or on some remote device). When the
25 trusted core is later loaded into memory 110, a component of computer 100 can

1 extract the cryptographic measure of the trusted core in memory 110 for
2 comparison to the stored cryptographic measure – if the two measures match then
3 it can be safely assumed that the trusted core has not been altered (e.g., by a
4 malicious user or program) while on device 118 or in memory 110.

5 In the illustrated example, the extracted cryptographic measure is stored in
6 digest 138. Digest 138 refers to one or more registers (or other storage
7 mechanisms) used to store a cryptographic measure of the trusted core as it is
8 loaded. One such measure is a “cryptographic digest” that is calculated using a
9 one-way hashing operation, such as SHA-1 (Secure Hash Algorithm 1). The
10 cryptographic digest has the property that it is extremely difficult to find a second
11 pre-image (in this case, a second trusted core) that when digested produces the
12 same hash value. Hence the digest-register contains a value that can be considered
13 to uniquely represent the trusted core in use. For a basic introduction of
14 cryptography, the reader is directed to a text written by Bruce Schneier and
15 entitled “Applied Cryptography: Protocols, Algorithms, and Source Code in C,”
16 published by John Wiley & Sons with copyright 1994 (or second edition with
17 copyright 1996).

18 Alternatively, measures other than a digest can be used. For example, the
19 logic could require that the trusted core be accompanied by a digital certificate(s).
20 The chipset could ensure that the certificate(s) matches the trusted core image, and
21 then save the certificate(s), a public key(s) obtained from the certificate(s), or a
22 digest(s) of the certificate(s) in the register.

23 Eventually, the computer is ready to load and initialize the component that
24 will provides the “root of trust” for the running computer. A significant problem
25 faced in previous computers is that all code that has executed beforehand can

The trusted core can operate in any of a wide variety of manners to make itself trustworthy. The manner in which the trusted core operates can vary, as can the amount of security or trustworthiness provided by the trusted core. Figs. 2 and 3 illustrate exemplary manners in which a trusted core can be implemented. Figs. 2 and 3, however, illustrate only examples of how the trusted core can be implemented, and the trusted core can actually be implemented in different manners. Generally, the trusted core can use whatever facilities are provided by the computer components to protect itself once allowed to execute, such as rings, curtaining, etc.

It should be noted that the invention described herein does not limit the nature of the code that will comprise the “root of trust.” It is anticipated that this code will take measures (such as programming the CPU memory controllers) to protect itself from code it might run, or devices it might program. However, the “root of trust” may be a full OS, a microkernel, a Hypervisor, or some smaller component that provides specific security services. Hereafter, we refer to such a component as the “trusted core.”

In Fig. 2, the trusted core is implemented by taking advantage of different privilege levels of CPUs 102, 104 of Fig. 1 (e.g., rings in an x86 architecture processor). In the illustrated example, these privilege levels are referred to as rings, although alternate implementations using different processor architectures

1 may use different nomenclature. The multiple rings provide a set of prioritized
2 levels that software can execute at, often including 4 levels (Rings 0, 1, 2, and 3).
3 Ring 0 is typically referred to as the most privileged ring. Software processes
4 executing in Ring 0 can typically access more features (e.g., instructions) than
5 processes executing in less privileged Rings. Furthermore, a processor executing
6 in a particular Ring cannot alter code or data in a higher priority ring. In the
7 illustrated example, a trusted core 160 executes in Ring 0, while an operating
8 system 162 executes in Ring 1 and applications execute in Ring 3. Thus, trusted
9 core 160 operates at a more privileged level and can control the execution of
10 operating system 162 from this level. Additionally, the code and/or data of trusted
11 core 160 (executing in Ring 0) cannot be altered directly by operating system 162
12 (executing in Ring 1) or applications 164 (executing in Ring 3). Rather, any such
13 alterations would have to be made by the operating system 162 or an application
14 164 requesting trusted core 160 to make the alteration (e.g., by sending a message
15 to trusted core 160, invoking a function of trusted core 160, etc.).

16 Alternatively, the operating system may be separated into a memory
17 manager component that operates as trusted core 160 with the rest of the operating
18 system operating as OS 162. The trusted core 160 then controls all page maps and
19 is thus able to shield trusted agents executing in Ring 3 from other components
20 (including OS 162). In this alternative, additional control also needs to be added
21 (e.g., in memory controller 106 of Fig. 1) to protect the trusted core from other
22 busmasters that do not obey ring privileges.

23 In Fig. 3, the trusted core is implemented by establishing two separate
24 "spaces" within computer 100: a trusted space 166 (also referred to as a protected
25 parallel area, or curtained memory) and a normal (untrusted) space 168. These

spaces can be, for example, one or more address ranges within computer 100. Both trusted space 166 and normal space 168 include a user space and a kernel space, with the trusted core 170 being implemented in the kernel space of trusted space 166. A variety of trusted applets, applications, and/or agents can execute within the user space of trusted space 166, under the control of trusted core 170. However, any application 174, operating system 176, or device driver 178 executing in normal space 168 is prevented, by trusted core 170, from accessing trusted space 166. Thus, no alterations can be made to applications or data in trusted space 166 unless approved by trusted core 170.

Returning to Fig. 1, the trusted core 146 is illustrated in three portions: a platform trusted core portion 148, a common trusted core portion 150, and a trusted core data portion 152. Although the portions 148, 150, and 152 are illustrated as being arranged contiguously in memory 110, the portions need not be so arranged. E.g., different portions can be stored in different non-contiguous areas, and different sections of each portion can be stored in different non-contiguous areas. Alternatively, depending on the manner in which trusted core 146 operates when executing, the portions may need to be arranged in physically contiguous locations within memory 110 (that is, actually stored in physically contiguous locations of the RAM, not being paged out to another device in accordance with virtual memory).

Trusted core data portion 152 is an area of memory 110 that is used to store data used by the trusted core. Platform trusted core portion 148 includes code that is specific to the particular hardware used in computer 100 (e.g., the specific type of CPU, the type of chipset, etc.). Common trusted core portion includes code that implements the operation and functionality of the trusted core but is not specific to

1 the particular hardware used in computer 100. By separating the two portions 148
2 and 150, the specifics of how interaction with the hardware of computer 100 (e.g.,
3 how to set a timer) can be abstracted to portion 150 by portion 148, so the same
4 common trusted core portion can be used on different computers. For example,
5 the common trusted core code 150 can operate as if all timers were set in the same
6 manner, while platform trusted core code 148 operates to convert the instructions
7 from code 150 into commands that are actually understood by the underlying
8 hardware of computer 100.

9 Trusted core 146 can be loaded into memory 110 from the same source or
10 alternatively different sources. By way of example, platform trusted core portion
11 148 may be loaded into memory 110 from the chipset of computer 100 (the chipset
12 refers to a set of one or more chips the provides the interfaces between the
13 computer's subsystems, including the buses and electronics to allow the CPUs,
14 memory, I/O devices, etc. to interact, and may include memory controller 106, I/O
15 controller 108, and buses 112 and 114), while the common trusted core portion
16 150 may be loaded into memory 110 from mass storage device 118. By way of
17 another example, one portion 148, 150, or 152 may be loaded into memory 110
18 from network adapter 128 (e.g., from a remote server), or the portions may all be
19 loaded from network adapter 128 but from different sources (e.g., different remote
20 servers). Additionally, different parts of one or more of portions 148, 150, and
21 152 may be loaded from different sources (e.g., part of portion 150 may be loaded
22 into memory 110 from mass storage device 118 and another part of portion 150
23 may be loaded from a remote server via network adapter 128).

24 Additionally, each portion may be generated by combining various overlaid
25 parts from one or more sources. For example, a first part could be read from one

1 source (e.g., a local disk) and a second part read from another source (e.g., a
2 remote server over the network), and then the two parts could be combined (e.g.,
3 exclusive-OR the bits of each part) to generate the portion.

4 It should be noted that the trusted core 146 need not be stored in a secure
5 location and/or loaded into memory 110 in a secure manner. The trusted core 146
6 may be stored on any publicly-accessible (or semi-publicly accessible) area, such
7 as mass storage device 118, a remote server accessed via network adapter 128, etc.
8 Rather, a cryptographic measure of trusted core 146 can be verified after loading
9 into memory 110 (and after it is protected so that no further modifications by
10 untrusted code are possible). The cryptographic measure can be generated in any
11 of a wide variety of conventional manners, and is designed so that any change in
12 the trusted core is reflected in the cryptographic measure. For example, if a
13 malicious user were to attempt to alter the trusted core by deleting or modifying an
14 instruction(s), or adding a new instruction(s), in an attempt to subvert the security
15 of the trusted core, that alteration would be reflected in the cryptographic measure
16 (so any subsequently generated cryptographic measure would not match a
17 previously generated cryptographic measure due to the change in the
18 instruction(s)). In one implementation, the cryptographic measure is obtained by
19 generating a digest value of the trusted core, such as in accordance with the well-
20 known SHA, SHA-1, MD5 (Message Digest 5), etc. algorithms.

21 This cryptographic measure is generated during the secure initialization
22 process described herein, and is made subsequently available to challengers (e.g.,
23 a bank, IT administrator, smart card, etc.). Interpreting the digest can be done in
24 several ways. It could be a "well known" value, such as a company publishing the
25 expected value of the its trusted core on its web site. Alternatively, the company

1 may generate a certificate for each trusted core it writes, and ship it with the
2 trusted core (or make it available on a web site). This allows challengers to see
3 the digest and ask for (or look for) a certificate that "names" the digest. It should
4 be noted that this certificate should not be in the core itself because the certificate
5 names the digest.

6 It should be noted that the initialization process described herein does not
7 limit what a user may try to run as the "trusted core" – the user can try to run
8 virtually any set of instructions as the trusted core. However, if the trusted core is
9 tampered with, then the instructions attempting to run as the "trusted core" (the
10 tampered with trusted core) will not be trusted by other parties because the
11 tampered with trusted core will have a different cryptographic measure and will
12 not have access to keys that are accessible to the non-tampered with trusted core.

13 It should also be noted that the initialization process described herein
14 naturally and easily allows a computer to run different trusted cores serially, at
15 arbitrary times, without re-booting the system. For example, the user can begin
16 executing trusted core *x*, then switch to executing trusted core *y* (which may be a
17 different version of the trusted core *x*, or an entirely different trusted core), and
18 then switch to trusted core *z* (or back to trusted core *x*), etc.

19 Memory controller 106 operates to ensure that trusted core 146, once
20 loaded into memory 110, can be initialized and begin execution in such a manner
21 that any previous state of computer 100 does not compromise the integrity of the
22 initialization process. It should be noted memory controller 106 operates to
23 provide a secure environment in which the trusted core can be initialized.
24 However, once the initialization process is completed, trusted core 146 is
25 responsible for its own security. Additionally, memory controller 106 does not

1 generate any guarantees or assurances regarding the trustworthiness or security
2 provided by trusted core 146.

3 Initialization of trusted core 146 refers to beginning execution of trusted
4 core 146. This initialization typically occurs after trusted core 146 has been
5 loaded in its entirety into memory 110, although alternative implementations may
6 allow for some parts of code 146 to be loaded into memory after the initialization
7 process occurs. The initialization process is performed by a set of trusted core
8 initialization instructions that are part of trusted core 146 (and maybe part of one
9 or more of portions 148, 150, and 152). The exact acts performed by the
10 initialization process can vary based on the specific manner in which the trusted
11 core operates as well as the specific hardware of computer 100 (e.g., the CPU
12 architecture), but generally include establishing the protection provided by the
13 trusted core (e.g., setting up a trusted space) and initializing CPUs to use this
14 protection. One of the things the "initialization code" must do is provide a "reset
15 vector" which is left in place, and is always ready to deal with, any processor that
16 is reset, initiated, hot-plug added, etc. By doing this, the trusted core ensures its
17 continued integrity in the face of profound changes in the CPU configuration of
18 the machine (e.g. having new ones added on the fly.)

19 Initialization of trusted core 146 begins in response to a "Trusted Core
20 Initialization" command issued by one of CPUs 102, 104. The Trusted Core
21 Initialization command is commonly issued during booting of computer 100,
22 although it may occur any time during or after computer 100 has been fully
23 booted. The Trusted Core Initialization command is designed to trigger
24 initialization of the trusted core, and can be implemented in any of a wide variety
25 of manners. For example, the command could be a write to a particular port, a

1 write to a particular address, etc., however, the port or address should have the
2 property that the trusted core can protect the port or address using whatever
3 protection means that hardware provides after the core has gained control. This is
4 to ensure that once a trusted core is running, it is not vulnerable to the simple
5 denial of service attack that any untrusted code can remove it from memory or
6 replace it. Alternatively, if such protection is not provided (e.g., the Trusted Core
7 Initialization command has to be invoked by some unprotectable port) then it may
8 be a "run once" operation which refuses to perform its function again until the
9 system has been rebooted. However, such a machine loses the ability to stop and
10 start trusted cores or to use more than one trusted core per boot.

11 In one implementation, the Trusted Core Initialization command includes
12 three parameters: start, length of code, and length of memory. The start parameter
13 refers to the location in memory 110 where trusted core 146 begins (e.g., the
14 memory address of the first instruction of platform trusted code portion 148).
15 Alternatively, especially in embodiments where trusted core 146 is not located in
16 physically contiguous memory, a memory descriptor list (MDL) can be identified
17 as the start parameter, identifying to memory controller 106 which memory pages
18 include the trusted core. The length of code parameter refers to the length (e.g.,
19 the number of bytes) of the code of trusted core 146 (e.g., the combination of
20 portions 148 and 150, but excluding trusted core data portion 152). The length of
21 memory parameter refers to the length (e.g., the number of bytes) of trusted core
22 (e.g., the combination of all three portions 148, 150, and 152). These parameters
23 are passed to memory controller 106 either prior to (or as part of) the Trusted Core
24 Initialization command. In one implementation, these parameters are loaded by
25

1 one of CPUs 102, 104 into parameter registers 153 of memory controller 106 prior
2 to issuance of the Trusted Core Initialization command.

3 Upon receipt of the Trusted Core Initialization command, memory
4 controller 106 protects memory 110 from all CPUs 102, 104 and any other bus
5 masters in computer 110. Given that CPUs 102, 104, as well as any other bus
6 masters in computer 100, cannot directly access memory 110, but rather have to
7 access memory 110 via memory controller 106, memory controller 106 can readily
8 protect memory 110 from the CPUs and bus masters. By protecting memory 110,
9 memory controller 106 ensures that other code (e.g., being executed by a CPU or
10 controlling another bus master) does not interfere with the trusted core
11 initialization process. In protecting memory 110, memory controller 106 initially
12 allows the only access to memory 110 to be the memory refresh signals from a
13 memory refresh controller (not shown) that allow DRAM to be refreshed and
14 maintain its state.

15 On a machine with factored memory – that is, multiple blocks of memory
16 with multiple memory controllers – multiple different approaches may be used. In
17 one approach, the multiple memory controllers coordinate with each other to apply
18 the Trusted Core Initialization operator across all memories. In another approach,
19 the memory-controller/memory selected to do the Trusted Core Initialization does
20 all of it (so the whole trusted core is loaded into one "bank") and other memory
21 controllers ignore it. Since all of the trusted core is in a single protected memory
22 bank, the activities of the others do not matter. The selected memory controller
23 still forces the behavior of all CPUs, so the platform hardware is set up to allow it
24 to do that.

Control logic 136 in memory controller 106 also sets a trusted core initialization bit 154 within a secure portion 156 of memory controller 106. This secure portion 156 is secure in that only control logic 136 is allowed to access the registers and/or memory within secure portion 156. Secure portion 156 may not be made visible to other components in computer 100 external to memory controller 106, or alternatively control logic 136 may simply ignore (or reject) any commands received at memory controller 106 attempting to access any register or memory within secure portion 156. The trusted core initialization bit 154 allows control logic 136 to know whether the trusted core initialization process is in-process, and the use of bit 154 is discussed in more detail below.

Memory controller 106 can protect memory 110 in a variety of different manners. In one implementation, memory controller 106 prevents CPUs 102, 104 from issuing read or write requests on processor bus 112, thereby preventing CPUs 102, 104 from accessing memory 110. The manner in which CPUs 102, 104 are kept off the bus varies based on the architecture of CPUs 102, 104 and processor bus 112 (e.g., in certain Intel-architecture implementations, CPUs can be prevented from issuing read and write requests on the bus by asserting a BNR# (Block Next Request) signal on processor bus 112, or by issuing a halt (e.g., HLT) command to the CPUs which halts the operation of each CPU until it is reset). CPUs 102, 104 will eventually be allowed to again issue read and write requests on processor bus 112, so control logic 136 also compares all requests that memory controller 106 receives from a CPU 102, 104 to the address range of trusted core 146 in memory 110 – any request that memory controller 106 receives from a CPU 102, 104 during this trusted core initialization process (e.g., with bit 154 set) that is within this address range is accepted and responded to in a conventional

1 manner, while any such request that is not within this address range or that is from
2 a component other than a CPU 102, 104 is rejected (e.g., memory controller 106
3 may refuse to acknowledge the request, may deny the request, etc.). Alternatively,
4 requests received that are not within this address range may be allowed, but with
5 the understanding that such areas of memory are not protected.

6 Memory controller 106 also prevents other bus masters on I/O bus 114 from
7 accessing memory 110. Additionally, memory controller 106 can prevent other
8 bus masters on I/O bus 114 from affecting cache memories (e.g., within a CPU or
9 on bus 112), such as by not issuing any such transactions onto bus 112. For
10 example, memory controller 106 may have I/O controller 108 assert a signal on
11 I/O bus 114 that prevents any of the bus masters from issuing read and write
12 requests on bus 114. By way of another example, memory controller 106 may
13 simply ignore any memory access requests it receives from I/O controller 108,
14 such as by having I/O interface 134 deny such requests, refuse to accept or
15 acknowledge such requests, etc. Although preventing access to memory 110 can
16 be implemented in similar fashions for both CPU accesses and I/O bus master
17 accesses, it should be noted that no I/O bus master accesses are allowed during the
18 initialization process even though CPU access are allowed.

19 Memory controller 106 may immediately protect memory 110 upon receipt
20 of the Trusted Core Initialization command, or alternatively wait for a period of
21 time or for some other action to occur before protecting memory 110. For
22 example, a particular bus master may be in the process of a long DMA transfer at
23 the time the Trusted Core Initialization command is received, and memory
24 controller 106 may opt to let the DMA transfer finish. Although the protection of
25

1 memory 110 may not be immediate, the trusted core initialization process does not
2 proceed until memory 110 is protected.

3 Once memory 110 is protected, control logic 136 generates a digest value
4 based on trusted core 146. Alternatively, another component (not shown), such as
5 a dedicated cryptographic processor, may be temporarily allowed access to
6 memory 110 in order to generate the digest value (or the digest could be held in
7 the cryptographic-processor, as long as the cryptographic-processor works
8 intimately with the memory controller to perform the digest operation only when
9 the CPU is performing the initialization process (and cannot be fooled by
10 untrusted software making this request of the cryptographic-processor)). This
11 access, however, may be limited, such as allowing the cryptographic processor to
12 only read from memory 110. This digest is generated using the same
13 cryptographic mechanism as discussed above. Control logic 136 then stores the
14 generated digest value as digest 138 in secure portion 156 of memory controller
15 106. Digest 138 can then be used for subsequent certification and/or
16 authentication purposes by trusted core 146.

17 Control logic 136 then maps the reset vector to a new reset vector, referred
18 to as the initialization vector, so that memory controller 106 responds to the next
19 read request for a particular address (e.g., FFFFFFFF₀₁₆) on processor bus 112 with
20 the initialization vector. In situations where the CPU always starts executing from
21 a known location (rather than using the indirection supplied by the reset-vector
22 fetch procedure), control logic 136 arranges to issue an instruction(s) as the first
23 one or few instructions that are fetched from the memory subsystem that causes
24 the CPU to begin executing instructions at the start of the trusted core (e.g., a
25 JUMP operation), or alternatively re-map the trusted core to appear at the fixed

processor reset vector. This initialization vector is the first address of trusted core 146. In one implementation, this mapping is accomplished by control logic 136 writing the start location of trusted core 146 (e.g., received as a parameter of the Trusted Core Initialization command) as reset vector 140. Whenever memory controller 106 receives a read request for the particular address (e.g., FFFFFFFF₀₁₆) on processor bus 112, control logic 136 checks bit 154 of secure portion 156. If the bit is set, then the mapped reset vector 140 is returned to the requesting CPU as the reset vector. However, if the bit is not set, then the initial BIOS boot block address is returned to the requesting CPU.

Control logic 136 then asserts a reset signal to a CPU on processor bus 112. In the situation where multiple CPUs are on processor bus 112, the reset signal is asserted to all of the CPUs (although alternatively the reset signal may be directed to a single CPU). The manner in which the reset signal is asserted (and the nature of the reset signal itself) can vary based on the architecture of CPUs 102, 104 and processor bus 112 (e.g., in certain Intel-architecture implementations, all CPUs on processor bus 112 can be reset by asserting a RESET# signal on processor bus 112).

Alternatively, other mechanisms can be used to reset the CPUs on processor bus 112. The goal of resetting a CPU is to clear the CPU of its state (e.g., all instructions and data that may reside within registers, caches, buffers, etc. of the CPU) so that that prior state cannot compromise the execution integrity of the trusted core initialization process. For example, so that instructions of a malicious application that are stored in a CPU's cache cannot compromise the integrity of the trusted core initialization process.

1 In addition to resetting the CPUs on processor bus 112, any other cache
2 memory or other device on processor bus 112 that includes memory is reset. For
3 example, an additional L3 cache (not shown) may also be coupled to processor bus
4 112, in which case control logic 136 would reset that L3 cache as well as the
5 CPUs. These additional cache memories or other devices on processor bus 112
6 can be reset in any of a variety of manners, such as writing random values to the
7 memories, writing a string of zeroes or ones to the memories, removing power
8 from the memories (e.g., temporarily not refreshing DRAMs), issuing specific bus
9 transactions to invalidate entire caches or specific cache-line entries, etc.

10 After resetting CPUs 102, 104 (and any other caches or devices on
11 processor bus 112 (except memory controller 106)), memory controller 106 allows
12 one of the CPUs on processor bus 112 to access memory 110. In the illustrated
13 example, any of the CPUs 102, 104 can be initially allowed to access memory 110
14 (although only one is allowed). Memory controller 106 can determine which of
15 multiple processors to allow to access memory 110 in any of a variety of manners,
16 such as some pre-determined setting, a random determination, allowing some
17 external (or CPU-internal) bus arbitration mechanism to determine, etc.

18 For the purposes of discussion, assume that memory controller 106
19 determines to allow CPU 102 to access memory 110. Memory controller 106 then
20 allows CPU 102 to access processor bus 112 (all other CPUs 104 are still
21 prevented from accessing processor bus 112). Once a CPU is reset, the CPU will
22 have no instructions (or at least no instructions or data that the CPU will read) in
23 any of its caches or buffers. CPU 102 will thus request instructions to execute by
24 asserting the particular address (e.g., FFFFFFFF₀₁₆) on processor bus 112. Memory
25 controller 106 receives this request, control logic 136 determines that trusted core

1 initialization process bit 154 is set, so memory controller 106 returns reset vector
2 140 to CPU 102. Also upon receipt of the particular address (e.g., FFFFFFFF₁₆)
3 control logic sets an additional per-processor vector bit 158. This bit 158 allows
4 control logic 136 to know, in the event another particular address (e.g.,
5 FFFFFFFF₁₆) is received, that reset vector 140 has already been returned to
6 another processor. The use of bit 158 is discussed in more detail below.

7 Upon receipt of reset vector 140, CPU 102 issues a read request on
8 processor bus 112 for the memory address in reset vector 140, which is the initial
9 instruction in trusted core 146 (and the initial instruction in the trusted core
10 initialization process). CPU 102 thus begins to execute the trusted core code 146.
11 Since CPU 102 was reset, no instructions (e.g., malicious code) previously loaded
12 into CPU 102 can affect its execution of trusted core 146. Additionally, since all
13 other bus masters are prevented from accessing memory 110, they cannot alter the
14 code 146 being fetched and executed by CPU 102 from memory 110.

15 CPU 102 continues to fetch and execute trusted core code 146, which
16 initializes the trusted core in computer 100. As discussed above, the exact manner
17 in which the trusted core is initialized can vary based on the architecture of
18 computer 100, CPUs 102, 104, the type of trusted core being implemented, etc.
19 As part of this initialization process, if trusted core 146 supports multiple
20 processors, an additional CPU start vector is loaded as reset vector 142. This
21 additional CPU start vector identifies the memory address in code 146 where
22 additional instructions are to be fetched in the event additional processors begin
23 executing instructions. This second reset vector can be used, for example, to have
24 additional processors begin executing instructions elsewhere in the trusted core
25 than the beginning of trusted core 146 (e.g., other than the beginning of the trusted

core initialization part of code 146). However, since no other CPU is allowed to run until the "End of Initialization" the vector at 142 won't be used until after "End of Initialization".

Eventually the trusted core initialization process completes, and an "End of Initialization" command is issued by CPU 102. The End of Initialization command can be asserted in any of a variety of manners, analogous to the Trusted Core Initialization command discussed above. Upon receipt of the End of Initialization command, memory controller 106 allows all remaining CPUs 104 on processor bus 112 to access memory 110. Each of these CPUs 104, having been reset, will have no instructions (or at least no valid instructions) in any of their caches or buffers. Each CPU 104 will thus request instructions to execute by asserting the particular address (e.g., FFFFFFFF₀₁₆) on processor bus 112. Memory controller 106 receives these requests, control logic 136 determines that additional processor vector bit 158 is set, so memory controller 106 returns reset vector 142 to each of CPUs 104.

Control logic 136 then clears the trusted core initialization bit 154, as the trusted core initialization process is complete. Memory controller 106 also ceases preventing other bus masters 110 from accessing memory 110. Thus, any additional protection against malicious code from another bus master is to be prevented by the execution of the trusted core 146, not memory controller 106.

Each CPU 104, upon receipt of reset vector 142, issues a read request on processor bus 112 for the memory address in reset vector 142, which is an instruction in trusted core 146. Each CPU 104 thus begins to execute instructions in the trusted core code 146. Since each CPU 104 was reset, no instructions (e.g.,

1 malicious code) previously loaded into the CPU 104 can affect its execution of
2 trusted core 146.

3 In one implementation, one or more of CPU 102, 104 may need microcode
4 loaded into it in order to function properly. This microcode may, for example, fix
5 errors or other bugs in the CPU, expose a different instruction set than the native
6 instruction set of the CPU, etc. This microcode can be included in trusted core
7 146 (e.g., in trusted core data portion 152) and instructions included in the trusted
8 core initialization process to load the microcode into the appropriate CPU(s).
9 Additionally, the authenticity of the microcode may be verified, such as by re-
10 computing a digest and comparing it to a certified digest within the microcode
11 (e.g., placed there by the author or distributor of the microcode) to ensure that the
12 microcode has not been altered since being signed by the distributor of the
13 microcode.

14 In one implementation, memory controller 106 provides the following
15 assurances in response to a Trusted Core Initialization command:

- 16 • All I/O access to memory stops. No component can access memory
17 except the refresh controller and CPUs (after being reset).
- 18 • The cryptographic measure of the trusted core is computed after I/O
19 accesses to memory stop.
- 20 • No CPU is able to re-start and access memory until after it has been
21 reset.
- 22 • All CPUs begin executing instructions at one of two locations in the
23 trusted core code, and that these locations are set from the trusted
24 core code.

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

1 In certain implementations, CPUs 102, 104 support two modes of
2 operation: a real mode and a protected mode. Real mode refers to offering the
3 programming environment of a predecessor CPU (e.g., the Intel 8086 processor),
4 while protected mode makes all instructions and architectural features of the CPU
5 available. In these implementations, CPUs 102, 104 are initialized into real mode
6 at power-on (and in response to a RESET# signal). The trusted core initialization
7 process begins, for each CPU, by transitioning that CPU into protected mode and
8 then turning memory paging on (memory paging refers to virtual memory, where
9 "pages" of memory can be referenced by the CPU and swapped between memory
10 110 and another storage device (e.g., device 118) as needed). Care should thus be
11 taken in the trusted core initialization process that for additional CPUs (after the
12 initial CPU) that are reset and permitted to access memory, they be able to run in
13 real mode for long enough to be transitioned to protected mode and paging turned
14 on. This can be accomplished, for example, by care in selection of the additional
15 processor start vector (reset vector 142).

16 Note also that if code external to the trusted core arranges a forced
17 processor reset (this is often possible programmatically, and is typically possible
18 on PC-class machines), then the processor will start up by executing the trusted
19 core, and not code that can potentially subvert a running core. In this case, the
20 core may choose to treat this as an exception condition, and destroy all existing
21 internal state (clear memory), or can take action to continue execution (for
22 example, reload the CPU protection profile that was in effect before the processor
23 reset).

24 It should be noted that the trusted core initialization process does not
25 require any additional bus transactions on processor bus 112, no additional pins on

1 CPUs 102, 104, or other such modifications to the CPUs 102, 104. Rather, the
2 initialization process is implemented in the memory controller (e.g., in the
3 chipset), thereby preventing the need for any modifications to the CPUs.

4 Although discussed herein as primarily performed during booting of a
5 computer, the trusted core initialization process can be performed at different
6 times. For example, the trusted core may not be loaded into memory 110 and the
7 Trusted Core Initialize command issued until an application that requires the
8 trusted core to operate is to be executed (e.g., the user is going to download
9 multimedia content that needs to be protected). Additionally, the trusted core
10 initialization process can be repeated multiple times. By way of example, the
11 trusted core code may have a "Terminate Trusted Core" command that terminates
12 execution of the trusted core. However, the trusted core could again be loaded into
13 memory 110 (if it were actually removed as part of the execution termination) at a
14 later time, and the trusted core initialization process repeated for the trusted core
15 code to begin executing again. And, multiple different trusted cores can be run,
16 one at a time, one after the other.

17 Additionally, it should be noted that after the trusted core initialization
18 process has completed and other CPUs are allowed to access memory, these
19 additional CPUs need not have been present in the computer when the trusted core
20 initialization process began. By way of example, the computer may support the
21 ability to "hot-plug" CPUs (add CPUs to the computer while the computer is
22 running). Any such newly added CPUs are allowed to access memory, starting
23 execution of instructions at the additional processor reset vector.

24 Fig. 1 illustrates an exemplary architecture where a single memory
25 controller 106 controls access to memory 110. Alternatively, the trusted core

1 initialization process may be implemented in architectures where there are
2 multiple memory controllers.

3 Fig. 4 illustrates an exemplary computer architecture employing distributed
4 memory controllers in accordance with certain embodiments of the invention.
5 Computer 200 is illustrated including multiple (*m*) CPUs 202, 204 coupled to a
6 processor bus 206. CPUs 202, 204 communicate with various I/O components
7 (not shown) via a bridge 208. Additionally, each CPU 202, 204 includes a
8 memory controller 210. The individual memory controllers 210 implement a
9 distributed memory controller architecture that performs analogous functions as
10 memory controller 106 of Fig. 1, except that the individual controllers 210 are
11 distributed across multiple CPUs. The distributed memory controller architecture
12 can be implemented in any of a wide variety of conventional manners, and may
13 include additional couplings (not shown) between the individual memory
14 controllers 210 to allow communications to be passed among the memory
15 controllers 210 (e.g., for synchronization of the individual memory controllers
16 210).

17 In the example of Fig. 4, each CPU includes its own directly attached
18 memory. The bus logic and protocols included in computer 200 ensure that each
19 CPU's view of the overall memory subsystem remains coherent. Such memory
20 architectures are often referred to as cache-coherent non-uniform-memory-access
21 systems (CC-NUMA). Alternatively, rather than each CPU having its own
22 attached memory, the memory controllers may be distributed among the multiple
23 CPUs, but access a single shared system memory.

24 Fig. 5 is a flowchart illustrating an exemplary process for a code
25 initialization process in accordance with certain embodiments of the invention.

1 The process of Fig. 5 is implemented by a computer, such as computer 100 of
2 Fig. 1 or computer 200 of Fig. 4, and may be implemented in software. The
3 process of Fig. 5 may be used for a trusted core initialization process, or
4 alternatively an initialization process for other code.

5 Initially, the normal boot process for the computer is started using the not
6 necessarily secure BIOS (act 252). Eventually, the code is loaded into memory
7 (act 254) and an Initialize Code command received (act 256). In response to the
8 Initialize Code command, memory is protected from all processors and bus
9 masters (act 258). A cryptographic measure of the code in memory is then
10 extracted and stored in a secure location (act 260). The processor reset vector is
11 then mapped to an initialization vector (act 262), such as the start of the code in
12 the memory.

13 Once the processor reset vector is mapped to the initialization vector, all the
14 processors are reset (act 264) and one of the processors is allowed to access
15 memory (act 266). The process then waits for the processor that is allowed to
16 access memory to execute a code initialization process from the memory (act 268).
17 The end of the code initialization process is signaled, for example, by a command
18 issued from the processor executing code initialization process. Once the code
19 initialization process is finished, the processor reset vector is re-mapped to an
20 additional processor start vector (act 270). Any other processors in the computer
21 are then allowed to access memory (act 272), as are any other bus masters in the
22 computer (act 274).

23 In the discussion herein, embodiments of the invention are described in the
24 general context of processor-executable instructions, such as program modules or
25 code, being executed by one or more conventional processors or controllers (e.g.,

control logic 136 of Fig. 1). Generally, program modules or code include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. In a distributed environment, program modules or code may be located in multiple memory storage devices.

Alternatively, embodiments of the invention can be implemented in hardware or a combination of hardware, software, and/or firmware. For example, all or part of the invention can be implemented in one or more application specific integrated circuits (ASICs) or programmable logic devices (PLDs).

Computing devices (such as device 100 of Fig. 1 or device 200 of Fig. 4) typically include at least some form of computer readable media. Computer readable media can be any available media that can be accessed by a computing device. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other media which can be used to store the desired information and which can be accessed by the computing device. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of

1 its characteristics set or changed in such a manner as to encode information in the
2 signal. By way of example, and not limitation, communication media includes
3 wired media such as wired network or direct-wired connection, and wireless media
4 such as acoustic, RF, infrared and other wireless media. Combinations of any of
5 the above should also be included within the scope of computer readable media.

6 For purposes of illustration, programs and other executable program
7 components such as the operating system are illustrated herein as discrete blocks,
8 although it is recognized that such programs and components reside at various
9 times in different storage components of the computer, and are executed by the
10 data processor(s) of the computer.

11 Conclusion

12 Although the description above uses language that is specific to structural
13 features and/or methodological acts, it is to be understood that the invention
14 defined in the appended claims is not limited to the specific features or acts
15 described. Rather, the specific features and acts are disclosed as exemplary forms
16 of implementing the invention.
17
18
19
20
21
22
23
24
25